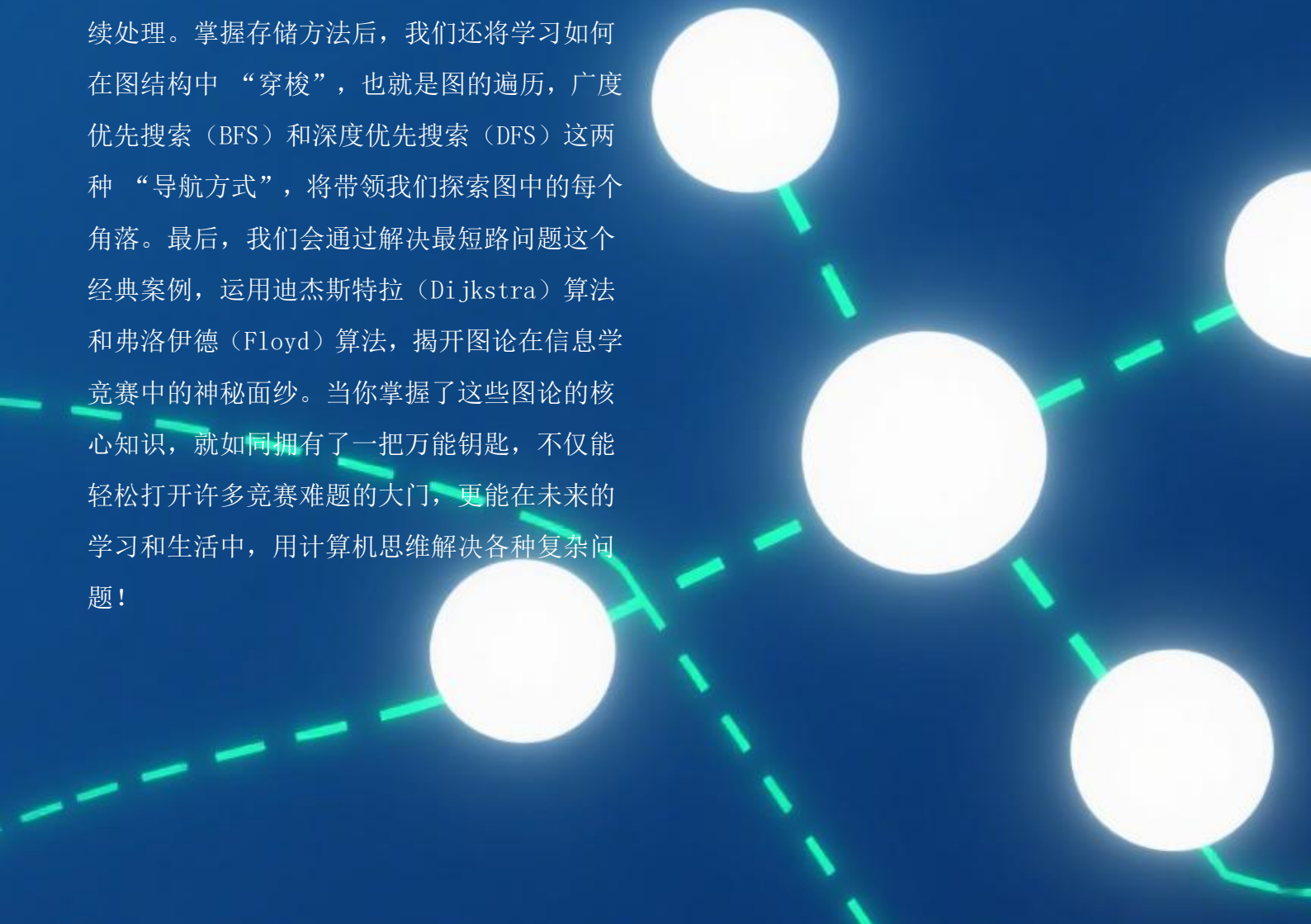
第四章
图论初步：存储与遍历

同学们，你是否曾好奇，外卖平台是如何在错综复杂的城市道路中，规划出骑手送餐的高效路线？网络游戏里，角色之间的社交关系网络又是怎样被构建和管理的？其实，这些生活中常见又有趣的现象背后，都藏着一个强大而实用的数学工具 —— 图论。在此之前，我们已经认识了数组、链表、树等数据结构，它们能帮助我们处理很多问题，但在面对更复杂的多对多关系时，就需要 “终极进化版” 的数据结构 —— 图结构登场了。

无论是像蜘蛛网般交织的城市间交通网络，每个城市作为一个顶点，连接城市的公路、铁路就是边；还是互联网中无数网页构成的链接世界，网页是顶点，超链接则是边，都可以抽象为由顶点和边组成的 “图”。而我们即将开启的这一章节，就像是一场探索图论奥秘的奇妙之旅。

在这段旅程中，我们首先会深入了解图的基本概念，认识顶点、边、权值这些构成图的 “基石”，区分无向图、有向图等不同类型图的特点。接着，我们要学习图在计算机中的存储方式，邻接矩阵和邻接表就像两种不同的 “收纳方法”，能帮助我们高效地把图的数据存储起来，以便后续处理。掌握存储方法后，我们还将学习如何在图结构中 “穿梭”，也就是图的遍历，广度优先搜索（BFS）和深度优先搜索（DFS）这两种 “导航方式”，将带领我们探索图中的每个角落。最后，我们会通过解决最短路问题这个经典案例，运用迪杰斯特拉（Dijkstra）算法和弗洛伊德（Floyd）算法，揭开图论在信息学竞赛中的神秘面纱。当你掌握了这些图论的核心知识，就如同拥有了一把万能钥匙，不仅能轻松打开许多竞赛难题的大门，更能在未来的学习和生活中，用计算机思维解决各种复杂问题！



4.1 图的基本概念

本节学习目标

- 理解图的定义与相关要素的基本概念，能够掌握图的结构。
- 掌握图的基本术语（如顶点、边、有向边、无向边、权等）及其实际意义。
- 识别多种图的类型（如连通图、强连通图等）的结构特点。
- 掌握图的不同路径（如欧拉路径、哈密顿路径等），并能够说明其定义和区别。
- 通过学习提升对图这一非线性结构的抽象思维与实际问题建模能力。

4.1.1 图的组成

图的定义

图（Graph）是一种用于表示对象之间关系的非线性数据结构，由**顶点（Vertex/Node）**和**边（Edge）**组成。顶点通常表示实体或对象，而边则表示这些实体或对象之间的关系。接下来我们将带领大家学习表示图基本结构的各种术语以及不同种类的边、图，一起来探索图的奥秘！

如图4.4.1，即为一个基本的图。ABCDEF为顶点，连接顶点的就是边。

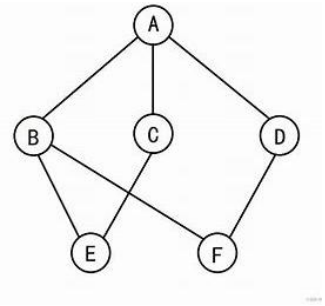


图4.1.1 基本的图

图的基本结构

1. 图由顶点和边组成：

（1）顶点（Vertex/Node）：顶点是图中的基本单元，表示图中的一个对象或实体。在图中，顶点通常用圆圈表示，并在圆圈内标注顶点的名称或编号。图4.4.1中圆圈A、B、C、D、E、F即为顶点。

（2）边（Edge）：边是连接两个顶点的连线，用于表示顶点之间的关系。

2. 边可以分为五种类型：

（1）无向边（Undirected Edge）：没有方向的边，表示两个顶点之间的双向连接。例如，无向边 (u,v) 表示顶点 u 和顶点 v 之间存在连接，且这种连接是双向的。图4.4.1中连接圆圈的没有箭头表示方向的连线即为无向边。

（2）有向边（Directed Edge）：带有方向的边，表示从一个顶点指向另一个顶点的单向连接。例如，有向边 u→v 表示从顶点 u 指向顶点 v 的连接。如图4.4.2中带有箭头的边。

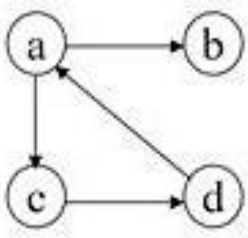


图4.1.2 有向边

（3）多重边（Multiple Edges）：两个顶点间存在多条边。边的数量用重表示。如图4.1.3中v1和v2两个顶点之间有两条边。

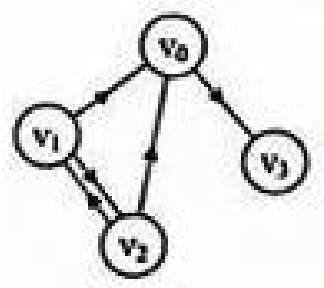


图4.1.3 多重边

（4）自环边（Loop）：顶点到自身的边。如图4.4.4中顶点0上的自环边。

（5）平行边（Parallel Edges）：两个顶点之间有两条无向边或有两条起点和终点相同的有向边。

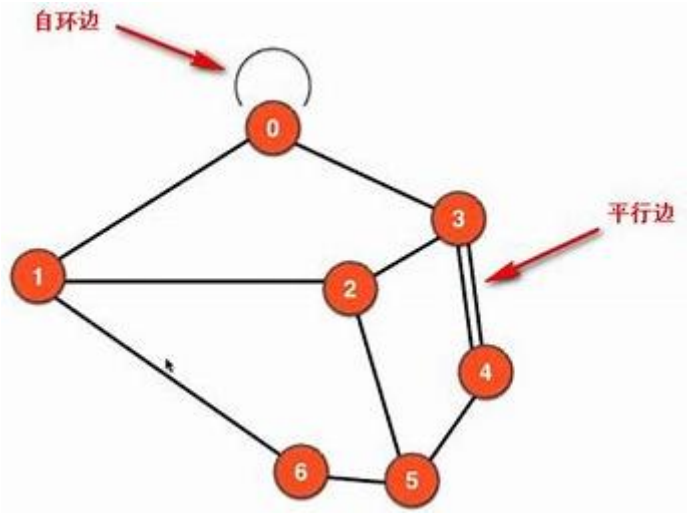


图4.1.4 复杂的图

4.1.2 图的分类

根据边的类型和方向分类

1. 无向图（Undirected Graph）：图中的所有边都是无向边，即边没有方向，顶点之间的连接是双向的。图4.4.1就是无向图。

2. 有向图（Directed Graph）：图中的所有边都是有向边，即边有方向，顶点之间的连接是单向的。图4.4.5就是有向图。大家注意进行区分，关注不同图的关键特征。

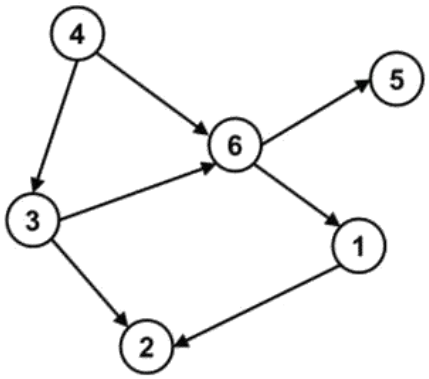


图4.1.5 有向图

3. 混合图（Mixed Graph）：图中同时包含有向边和无向边。这种图在某些复杂的应用场景中可能会出现，例如在交通网络中，某些路段可能是双向通行的（无向边），而某些路段可能是单向通行的（有向边）。

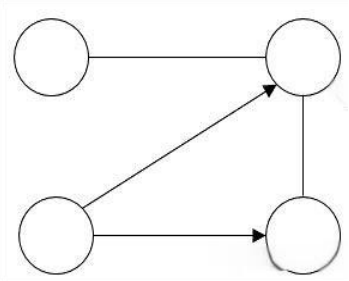


图4.1.6 混合图

拓展知识：特殊类型的图

完全图

在无向图中，每对顶点之间都有一条边相连；在有向图中，每对顶点之间都有一条双向边相连，这样的图被称为**完全图（Complete Graph）**。

思考：当完全图的顶点数为 n 时，边数为多少呢?相信大家都计算出来了。在无向图中有n(n-1)/2条边，在有向图中则有n(n-1)条边。

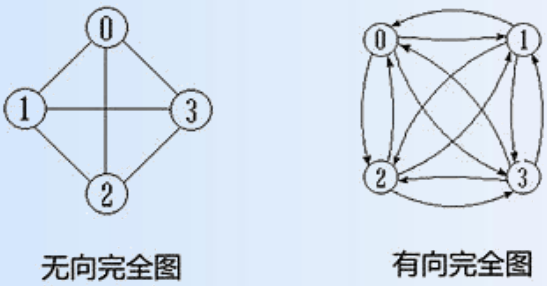


图4.1.7 完全图

顶点的度数

1. 度数（Degree）：仅在无向图中适用，无向图顶点的度数是指与该顶点相连的边的数量。例如，如果一个顶点与三条边相连，那么它的度数为 3。

2. 入度（In-Degree）和出度（Out-Degree）：仅在有向图中适用，有向图中入度是指指向该顶点的边的数量，出度是指从该顶点出发的边的数量。例如，对于顶点 v，如果有三条边指向 v，而从 v 出发的边有两条，那么 v 的入度为 3，出度为 2。

4.1.3 图的路径与连通性

在图的理论中，路径和连通性是两个非常重要的概念。它们不仅描述了图中顶点之间的连接关系，还为许多图算法提供了基础。本小节将详细介绍这些概念及其相关术语。

连通性

连通性（Connectivity）是图的一个基本性质，用于描述图中顶点之间的可达性。

1. 连通图（Connected Graph）：在无向图中，如果任意两个顶点之间都存在路径，则称该图为连通图。路径是指从一个顶点到另一个顶点的顶点序列，序列中的相邻顶点通过边相连。

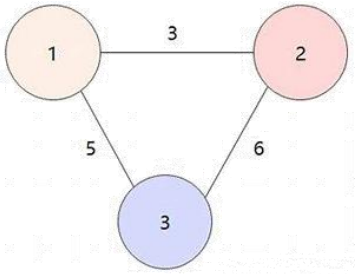


图4.1.8 连通图

2. 强连通图（Strongly Connected Graph）：在有向图中，如果任意两个顶点u和v之间都存在从u到v和从v到u的路径，则称该图为强连通图。这意味着在强连通图中，任意两个顶点之间都可以双向到达。

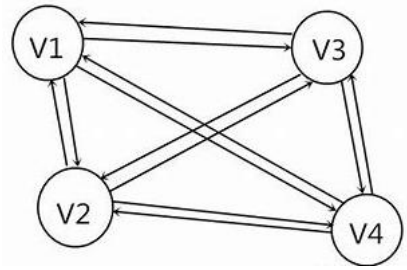


图4.1.9 强连通图

路径与回路

路径和回路是图中描述顶点之间连接关系的重要概念。

1. 路径（Path）： 路径是从一个顶点到另一个顶点的顶点序列，序列中的相邻顶点通过边相连。路径的长度是指路径中边的数量。例如，在图4.1.10的图中，路径 B→D→F 表示从顶点 B 经过顶点 D 到达顶点 F 的一条路径。

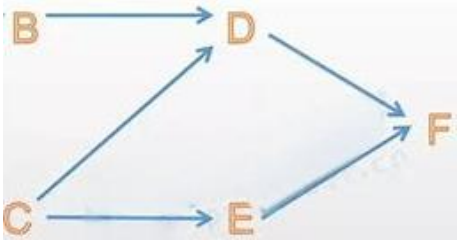


图4.1.10 路径示意图

2. 简单路径（Simple Path）： 简单路径是指路径中顶点不重复的路径。也就是说，路径中的每个顶点（除了起点和终点）都只出现一次。例如，路径 B→D→F 是简单路径，而路径 B→D→B→D→F 不是简单路径，因为顶点B和D重复出现了。

3. 回路（Cycle）： 回路是指起终点相同的路径。也就是说，路径的起点和终点是同一个顶点。例如，路径 v0→v2→v3→v0 是一个回路。

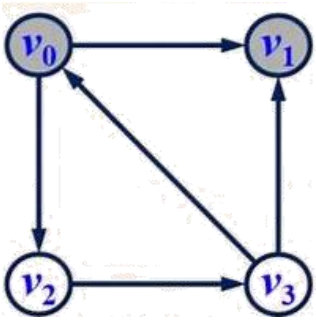


图4.1.11 回路示意图

4. 简单回路（Simple Cycle）： 简单回路是指除起点和终点外，路径中顶点不重复的回路。例如在图4.1.11中，路径 v0→v2→v3→v0 是简单回路，而路径 v0→v2→v3→v0→v2→v3→v0 不是简单回路，因为顶点v2和v3重复出现了。

特殊路径

1. 欧拉路径（Eulerian Path）： 欧拉路径是指在图中遍历每条边恰好一次的路径。如果一个图存在欧拉路径，那么这个图称为欧拉图。欧拉路径的起点和终点可以不同。

图4.1.12中每条边上的序号为遍历的顺序，在图中遍历每条边恰好一次。

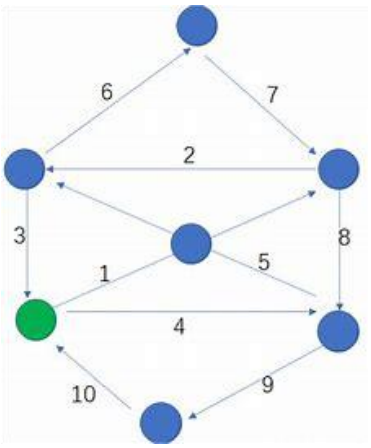


图4.1.12 欧拉图

2. 哈密顿路径（Hamiltonian Path）：哈密顿路径是指在图中遍历每个顶点恰好一次的路径。如果一个图存在哈密顿路径，那么这个图称为哈密顿图。哈密顿路径的起点和终点可以不同。

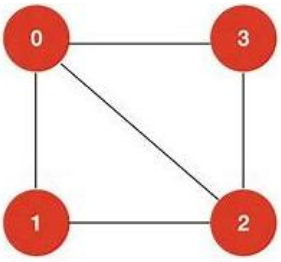


图4.1.13 哈密顿图

知识链接：边权重

边权重是边的数值属性，通常用于表示边的某种特性，用于最短路径或最小生成树。请大家回忆之前的知识，想一想如何运用呢？

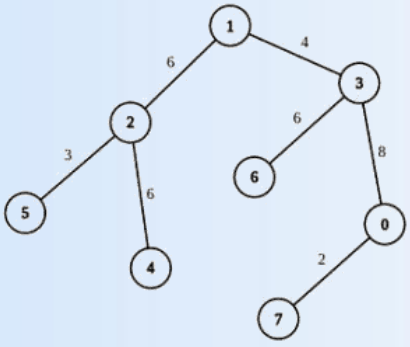
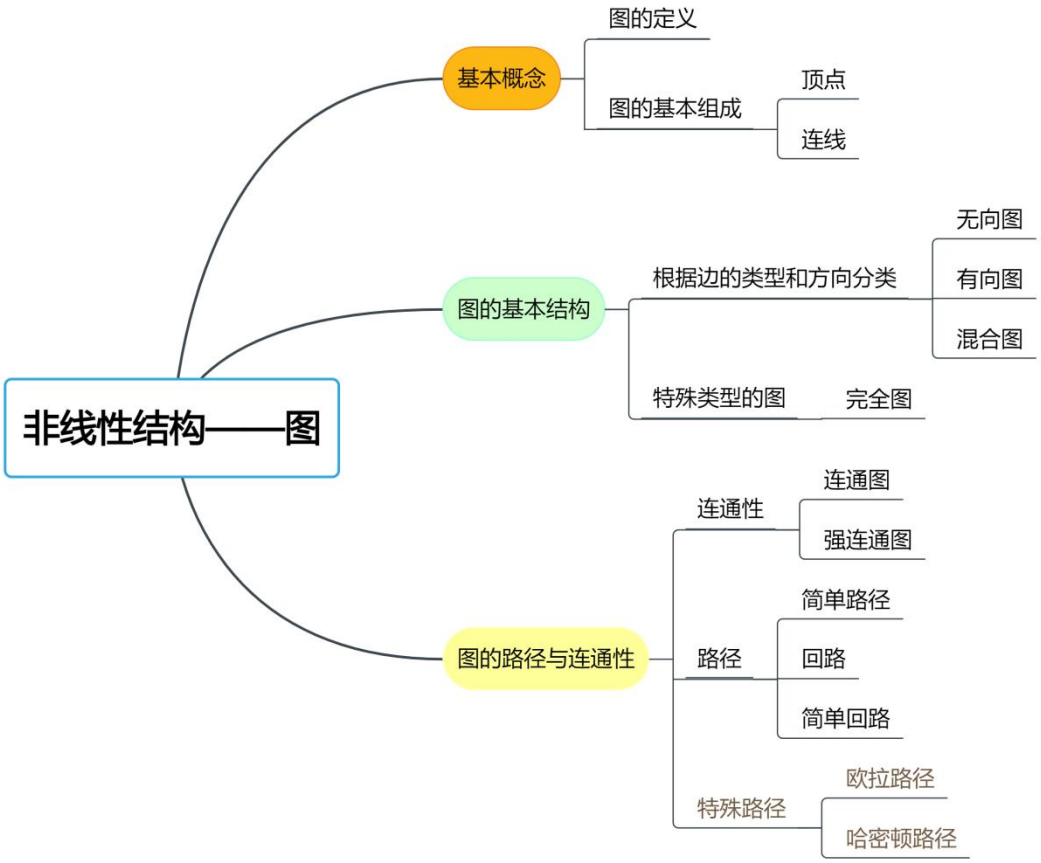


图4.1.14 边权重

知识图谱



练习提升

一、判断题

1. 有向图中，如果从顶点 u 到顶点 v 有路径，那么从顶点 v 到顶点 u 一定有路径。（ ）

2. 无向图中，如果两个顶点之间有边，那么这两个顶点一定连通。（ ）

3. 欧拉路径是指在图中遍历每条边恰好一次的路径。（ ）

4. 哈密顿路径是指在图中遍历每条边恰好一次的路径。（ ）

二、选择题

1. 下列关于图的说法中，正确的是（ ）。

A. 有向图中，如果从顶点 u 到顶点 v 有路径，那么从顶点 v 到顶点 u 一定有路径。

B. 无向图中，如果两个顶点之间有边，那么这两个顶点一定连通。

C. 有向图中，如果从顶点 u 到顶点 v 有路径，那么从顶点 v 到顶点 u 一定没有路径。

D. 无向图中，如果两个顶点之间没有边，那么这两个顶点一定不连通。

2. 下列关于图的路径的说法中，错误的是（ ）。

A. 简单路径是指路径中顶点重复的路径。

B. 回路是指起终点相同的路径。

C. 欧拉路径是指在图中遍历每条边恰好一次的路径。

D. 哈密顿路径是指在图中遍历每个顶点恰好一次的路径。

三、计算题

1. 给定一个无向图，顶点集合为 V={A,B,C,D,E}，边集合为 E={(A,B),(A,C),(B,C),(B,D),(C,D),(D,E)}。判断该图是否为连通图。

2. 给定一个有向图，顶点集合为 V={A,B,C,D}，边集合为 E={(A,B),(B,C),(C,D),(D,A),(A,C)}。判断该图是否为强连通图。

四、应用题

1. 在一个交通网络中，有5个城市，分别用顶点 A,B,C,D,E 表示。城市之间的道路用边表示，道路的长度用边的权重表示。给定边集合和权重如下：

E={(A,B,10),(A,C,15),(B,C,20),(B,D,25),(C,D,30),(D,E,35)}。

问题：从城市 A 到城市 E 的最短路径是什么？

2. 在一个社交网络中，有4个用户，分别用顶点 A,B,C,D 表示。用户之间的关系用边表示，给定边集合如下：

E={(A,B),(A,C),(B,C),(B,D),(C,D)}。

问题：该社交网络是否为连通图？为什么？

4.2 图的存储与遍历

本节学习目标

- 理解图的四种存储结构（边集、邻接表、邻接矩阵、链式前向星）的实现原理与适用场景。
- 掌握邻接表与邻接矩阵的核心操作时间复杂度对比，能针对“判断边存在性”“遍历邻接点”等操作，结合图结构特点选择最优存储方式。
- 熟练运用BFS和DFS算法完成图的遍历，能用队列实现BFS、递归/栈实现DFS，理解二者在最短路径、连通性问题中的应用差异。

4.2.1 图的存储方法

边集存储

边集存储（Edge List）是最直观、最基础的存储方式，它将图中的每一条边都单独记录下来。对于一个拥有E条边的图，可使用数组edges[E][2]（适用于无向图）来存储边信息，其中edges[i][0]和edges[i][1]分别表示第i条边连接的两个顶点。以一个简单无向图为例：

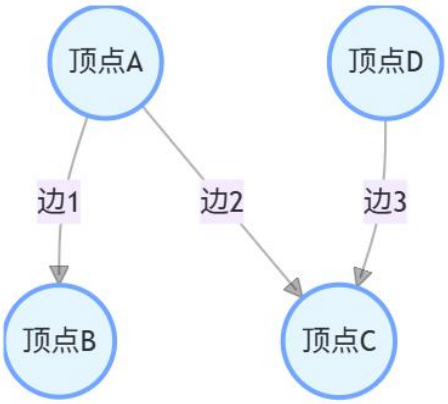


图4.2.1 简单的无向图

在上述图中，若采用边集存储，对应的数组内容为[[A, B], [A, C], [D, C]]。

尽管边集存储方式简单直接，但在实际竞赛应用中却并不常用。这是因为当我们需要查询某个顶点的所有相邻顶点时，需要遍历整个边集数组，时间复杂度高达O(E)。例如，若要查找顶点A的邻接点，需逐个检查数组中的每一条边，效率较低，因此通常仅作为理解图存储的基础概念。

邻接表存储

邻接表（Adjacency List）是竞赛中广泛使用的存储方式，它使用vector <vector<int>>来存储边信息。形象地说，邻接表为图中的每个顶点建立了一个“邻居清单”。外层容器对应图中的各个顶点，内层容器则记录与该顶点直接相连的其他顶点。

例如对于顶点A，其内层容器会存储所有与A有边相连的顶点。

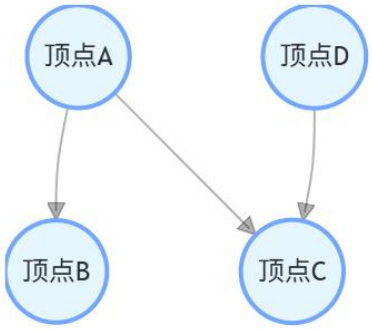


图4.2.2 邻接表示意图

以图4.2.2为例，使用 C++ 实现邻接表存储的代码如下：

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int N = 100;   // 最大顶点数
vector<int> adj[N];   // 邻接表

void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);   // 无向图，双向添加
}
```

图4.2.3 邻接表存储代码（上）

```
int main() {
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 2);

    // 输出顶点0的邻接点
    cout << "顶点0的邻接点: ";
    for (int i = 0; i < adj[0].size(); i++) {
        cout << adj[0][i] << " ";
    }
    return 0;
}
```

图4.2.4 邻接表存储代码（下）

邻接表存储方式在处理稀疏图（边数远小于顶点数平方的图）时具有显著优势，其空间复杂度为O(V + E)，仅存储实际存在的边，避免了空间浪费。同时，遍历某个顶点的所有邻接点时，只需访问其对应的内层容器，时间复杂度为O(E)，效率较高。

邻接矩阵存储

邻接矩阵（Adjacency Matrix）采用二维数组来存储图结构。假设图中有v个顶点，则创建一个二维数组graph[V][V]。对于无向图，若顶点i和顶点j之间存在边，则将graph[i][j]和graph[j][i]设为1（也可设为其他表示边存在的值）；若不存在边，则设为0。例如，对于一个包含3个顶点的无向图，若顶点0和顶点1有边相连，其邻接矩阵如下：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

图4.2.5 邻接矩阵

使用 C++ 实现邻接矩阵存储的代码如下：

```cpp
#include <iostream>
using namespace std;

const int N = 100;   // 最大顶点数
int graph[N][N];   // 邻接矩阵

void addEdge(int u, int v) {
    graph[u][v] = 1;
    graph[v][u] = 1;   // 无向图，双向标记
}

int main() {
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 2);

    // 输出邻接矩阵
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

图4.2.6 邻接矩阵存储代码

邻接矩阵的最大优点在于判断两点之间是否存在边极为快速，只需直接访问对应数组元素，时间复杂度为0。然而，其缺点也十分明显，无论图的稀疏程度如何，都需要占用O(VÂ²)的空间，当处理稀疏图时，会造成大量空间浪费。

链式前向星

链式前向星（Linked Forward Star）是一种基于链表思想、通过数组模拟链表实现的邻接表变体，常用于高效存储和处理图结构，尤其在需要对边进行特殊操作时表现出色。

在链式前向星中，通常需要定义以下几个关键数组：

- **head[i]**：表示以顶点\(i\)为起点的第一条边在边数组edge中的存储位置。

- **edge[i].to**：表示第\(i\)条边的终点。
- **edge[i].next**：表示与第\(i\)条边同起点的下一条边在边数组edge中的存储位置。
- **cnt**：用于记录当前边的数量。

以下是一个简单的链式前向星存储的 C++ 代码示例：

```cpp
#include <iostream>
using namespace std;

const int N = 100, M = 1000;  // 最大顶点数和最大边数
int head[N], cnt;
struct Edge {
    int to, next;
} edge[M];

void addEdge(int u, int v) {
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

int main() {
    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 2);

    // 遍历顶点0的邻接边
    for (int i = head[0]; i != -1; i = edge[i].next) {
        int v = edge[i].to;
        cout << v << " ";
    }
    return 0;
}
```

图4.2.7 链式前向星存储代码

还是以图二为例，其链式前向星存储结构可通过如下图示展示。在该图示中，顶点A有两条出边，head[A]指向第一条边E1（在edge数组中位置为0），E1的next指向同起点的第二条边E2（位置为1），E2的next为−1表示无后续同起点边；顶点D的出边情况同理。通过这种结构，能够高效地遍历每个顶点的邻接边。

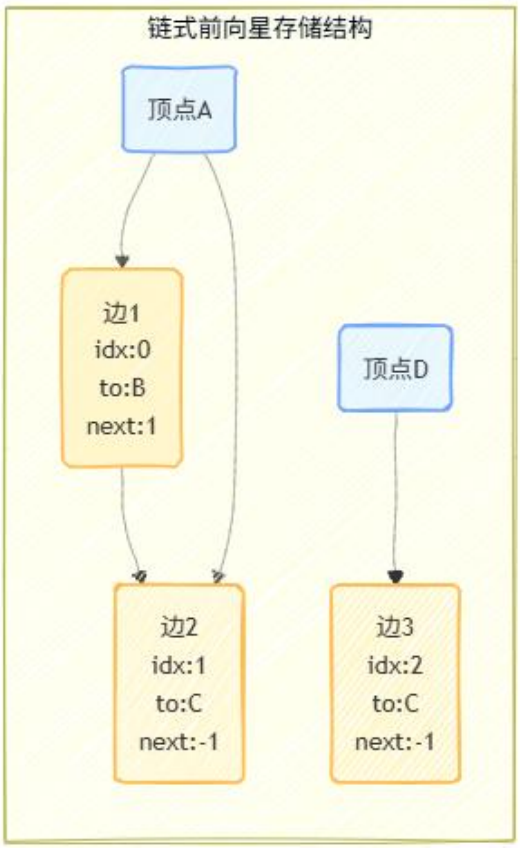


图4.2.8 链式前向星存储结构示意图

4.2.2 几种存储方法的比较

为了更全面、深入地理解不同图存储方法的特性，我们从存储结构、时间复杂度、空间复杂度和优缺点等多个维度进行详细对比。

存储结构对比

方法	实现	结构特点
邻接表	vector（C++）	为每个顶点构建邻接顶点列表，动态存储实际存在的边
邻接矩阵	二维数组	使用固定大小的方阵，通过矩阵元素标记顶点间连接关系
边集	线性列表或数组	简单记录图中每一条边的两个端点
链式前向星	多个数组（head数组、edge结构体数组等）	通过数组模拟链表，存储边的详细信息及连接关系

表4-1 存储结构对比表

时间复杂度对比

（注意：E指边数，V指节点数）

操作	邻接表	邻接矩阵	边集	链式前向星	分析
判断边存在性	O(E)	O(1)	O(E)	O(E)	邻接表和链式前向星需遍历邻接边查找；邻接矩阵直接访问对应元素；边集需遍历所有边判断
遍历邻接点	O(E)	O(V)	O(E)	O(E)	邻接表和链式前向星遍历邻接边；邻接矩阵遍历对应行；边集需遍历所有边筛选
添加边	O(1)	O(1)	O(1)	O(1)	均为简单的插入或标记操作

表4-2 时间复杂度对比表

空间复杂度对比

方法	空间复杂度	说明	适用场景
邻接表	O(V + E)	仅存储实际边和顶点信息，动态分配	稀疏图
邻接矩阵	O(VÂ²)	固定大小数组，无论图疏密均占用相同空间	稠密图
边集	O(E)	仅存储边信息，不记录顶点间接关系	需全局边操作场景
链式前向星	O(V + E)	与邻接表类似，但需额外存储边连接关系	需高效边操作场景

表4-3 空间复杂度对比表

优缺点对比

方法	优点	缺点
邻接表	空间利用率高，适合稀疏图；遍历邻接点效率高；动态存储灵活适应图规模变化	动态分配存在一定开销；判断边存在性需遍历邻接边，效率较低
邻接矩阵	判断边存在性极为快速；矩阵操作方便，易于实现	空间浪费严重，尤其在稀疏图中；占用固定空间，无法动态调整

边集	结构简单，易于实现；适合对全局边进行统一操作，如排序	遍历邻接点需遍历整个边集，效率低下；无法快速获取顶点邻接信息
链式前向星	空间利用率高，存储紧凑；适合对边进行特殊操作和处理	原理相对复杂，实现难度较高；代码调试和维护成本较大

表4-4 优缺点对比表

4.2.3 图的遍历

完成图的存储后，图的遍历操作是解决众多图论问题的核心环节。其中，**广度优先搜索（BFS）**和**深度优先搜索（DFS）**是最常用的两种遍历策略。下面以常用的邻接表存储为例，详细介绍这两种遍历方式。

广度优先搜索（BFS）

BFS 的遍历过程如同水波扩散，从起始顶点开始，逐层向外访问顶点。首先访问起始顶点的所有直接邻居，然后依次访问这些邻居的未访问邻居，以此类推，直到遍历完所有可达顶点。在实现 BFS 时，通常借助队列来管理待访问的顶点，这是因为队列先进先出的特性，恰好符合 BFS 逐层探索的逻辑。以如下简单图为例：

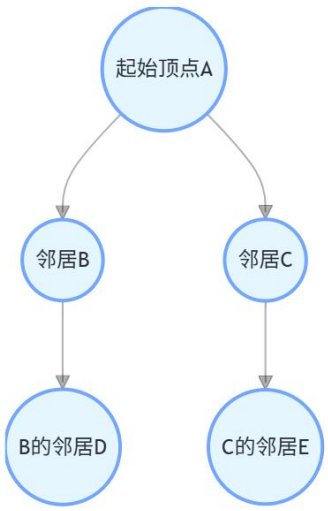


图4.2.9 广度优先搜索示意图

BFS 在很多场景中都有重要应用。例如，在寻找无权图中两个顶点的最短路径时，BFS 能够保证找到的路径是经过边数最少的路径；在求解迷宫问题时，BFS 可以按照离起点由近到远的顺序探索迷宫，找到从起点到终点的最短路线。

使用 C++ 实现 BFS 的代码如下：

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

const int N = 100;
vector<int> adj[N];
bool visited[N];

void bfs(int start) {
    queue<int> q;
    q.push(start);
    visited[start] = true;

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cout << u << " ";

        for (int i = 0; i < adj[u].size(); i++) {
            int v = adj[u][i];
            if (!visited[v]) {
                q.push(v);
                visited[v] = true;
            }
        }
    }
}
```

图4.2.10 BFS代码

深度优先搜索（DFS）

DFS 的遍历方式更像是一场不断深入的冒险，从起始顶点开始，沿着一条路径尽可能地深入访问相邻顶点，直到无法继续前进（即当前顶点的所有邻接顶点都已被访问），然后回溯到上一个顶点，继续探索其他未访问的路径，直至遍历完所有可达顶点。DFS 可以使用递归或者栈来实现，递归实现代码简洁直观，而栈实现则更便于理解其底层逻辑。

同样以之前的简单图为例，使用 C++ 递归实现 DFS 的代码如下：

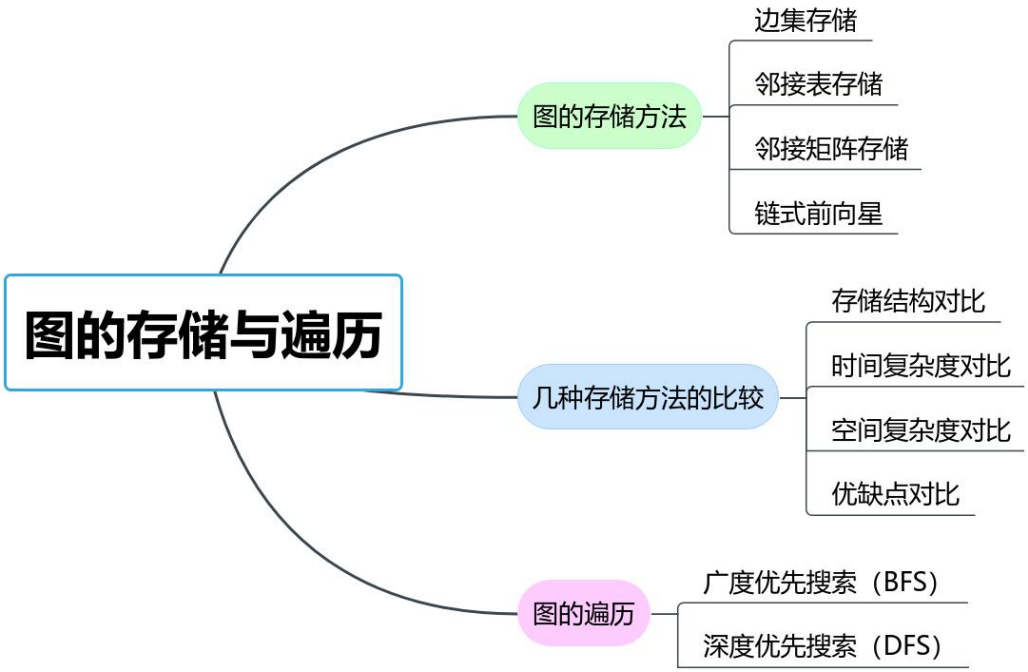
```cpp
#include <iostream>
#include <vector>
using namespace std;

const int N = 100;  // 定义最大顶点数
vector<int> adj[N];  // 邻接表存储图结构
bool visited[N];  // 标记数组，用于记录顶点是否已被访问

void dfs(int u) {
    visited[u] = true;  // 标记当前顶点已访问
    cout << u << " ";  // 输出当前访问的顶点

    for (int i = 0; i < adj[u].size(); i++) {  // 遍历当前顶点的所有邻接顶点
        int v = adj[u][i];  // 获取邻接顶点
        if (!visited[v]) {  // 如果邻接顶点未被访问
            dfs(v);  // 递归访问该邻接顶点
        }
    }
}
```

图4.2.11 DFS代码

DFS 在拓扑排序、寻找图的连通分量等问题中发挥着重要作用。例如，在有向无环图的拓扑排序中，通过 DFS 可以方便地确定顶点的先后顺序；在判断一个图是否连通时，从任意一个顶点开始进行 DFS，如果能够访问到所有顶点，则图是连通的，否则图存在多个连通分量。

通过对 BFS 和 DFS 的详细介绍，我们可以根据具体问题的需求和特点，选择合适的遍历方式，从而高效地解决各类图论问题。在后续的学习中，我们还会结合更多实际竞赛题目，深入探讨它们的应用技巧和优化方法。

知识图谱



练习提升

一、基础概念题

1. 比较邻接矩阵和邻接表的优缺点，并分别说明它们适用的场景（如稠密图、稀疏图）。

2. 给定一个图，顶点数为V，边数为E。请填写以下操作在不同存储方式下的时间复杂度：

操作	邻接表	邻接矩阵	边集	链式前向星
判断顶点u和v是否邻接				
遍历顶点u的所有邻接点				
添加一条边u→v				

二、代码实现题

3. 邻接表的BFS遍历

使用C++实现邻接表存储的无向图，并从顶点0出发进行BFS遍历，输出访问顺序。

输入示例：顶点数 V=4，边集为 {{0,1}, {0,2}, {2,3}}。

输出示例：0 1 2 3

4. 链式前向星的DFS遍历

使用链式前向星存储有向图，并实现非递归（栈）的DFS遍历，从顶点0出发输出访问顺序。

输入示例：顶点数 V=3，边集为 {{0,1}, {0,2}, {2,1}}。

输出示例：0 2 1（依赖遍历顺序）

三、综合应用题

5. 最短路径问题

给定一个无权无向图（边权为1），使用邻接表存储，编写BFS算法求解从顶点 s 到顶点 t 的最短路径长度。若不可达返回-1。

输入示例：

V=5, E=6, edges={{0,1}, {0,2}, {1,3}, {2,3}, {2,4}, {3,4}}

s=0, t=4

输出：3（路径0→2→4）

6. 图的连通分量计数

使用邻接矩阵存储图，通过DFS遍历统计图中的连通分量数量。

输入示例：

V=5, edges={{0,1}, {1,2}, {3,4}} // 图分为两个连通分量

输出：2

四、进阶思考题

7. 存储方式选择与优化

假设需要频繁进行两种操作：(1) 判断任意两顶点是否邻接；(2) 遍历某个顶点的所有邻接点。

若图的顶点数 V=1000，边数 E=3000，你会选择哪种存储方式？为什么？如果边数变为 E=800000，选择会变化吗？

4.3 图上问题案例——最短路

本节学习目标

- 理解Dijkstra算法、Bellman - Ford算法、Floyd - Warshall算法这三种算法解决图的最短路径问题的核心原理，熟知各算法的特性。
- 能够独立用 C++实现三种算法，学会调试代码以解决可能出现的逻辑错误、运行错误，了解算法性能优化的方向理解并查集的时间复杂度。
- 通过学习这三种算法，培养贪心、动态规划的算法思维，遇到问题能选择合适策略，学会分析算法的时间和空间复杂度。

4.3.1 单源最短路径算法——Dijkstra算法

算法简介

Dijkstra算法是由荷兰计算机科学家狄克斯特拉于1959年提出的，因此又叫狄杰斯特拉算法。是从一个顶点到其余各顶点的最短路径算法，解决的是有权图中最短路径问题。该算法主要特点是从起始点开始，采用贪心算法的策略，每次遍历到始点距离最近且未访问过的顶点的邻接节点，直到扩展到终点为止。普通的 Dijkstra 算法在处理大规模图时，由于每次都需要遍历所有未确定最短路径的顶点来找到距离源点最近的顶点，时间复杂度较高，为 O(V^2)，其中 V 是图中顶点的数量。在竞赛中，当图的规模较大时，这种复杂度可能会导致程序超时。因此，我们需要一种更高效的方法来优化 Dijkstra 算法，这就是**基于堆优化的 Dijkstra 算法**。

而基于堆优化的 Dijkstra 算法引入了优先队列（通常使用小顶堆实现）来优化选择最近顶点的过程。其核心思想是使用小顶堆来代替普通 Dijkstra 算法中的线性查找，从而将每次查找距离源点最近的顶点的时间复杂度从 O(V) 降低到 O(logV)。

算法详解

1. 初始化

创建一个数组 dist，用于记录源点到每个顶点的最短距离。将源点的距离初始化为 0，即 dist[s] = 0，其他顶点的距离初始化为无穷大。

创建一个布尔类型的数组 visited，用于标记每个顶点是否已经确定了最短路径。初始时，所有顶点的标记都设为 false。

创建一个小顶堆 pq，把源点及其距离（0）作为一个元素插入到堆中。堆中的元素通常是一个二元组 (distance, vertex)，按照距离从小到大排序。

2. 迭代过程

从优先队列 pq 中取出距离最小的顶点 u。若该顶点已被标记为已访问（即 visited[u] == true），则跳过该顶点，继续从堆中取出下一个元素；否则，将该顶点标记为已访问，即 visited[u] = true。

对于顶点 u 的所有邻接顶点 v，计算通过 u 到达 v 的新距离 new_dist = dist[u] + weight(u, v)，其中 weight(u, v) 是边 (u, v) 的权值。若 new_dist 小于当前记录的 dist[v]，则更新 dist[v] = new_dist，并将 (new_dist, v) 插入到优先队列 pq 中。

3. 终止条件

当优先队列为空时，算法结束，此时 dist 数组中存储的就是源点到每个顶点的最短距离。

接下来我们通过一道例题来详细学习基于堆优化的 Dijkstra 算法。

例题分析

【题目】

给定一个 n 个点，m 条有向边的带非负权图，请你计算从 s 出发，到每个点的距离。数据保证你能从 s 出发到任意点。

【输入格式】

第一行为三个正整数 n,m,s。第二行起 m 行，每行三个非负整数 ui,vi,wi，表示从 ui到 vi有一条权值为 wi的有向边。

【输出格式】

输出一行 n 个空格分隔的非负整数，表示 s 到每个点的距离。

【分步解析】

1. 定义边的结构体和无穷大常量：Edge 结构体用于表示图中的边，包含两个成员变量：to 表示边的终点，weight 表示边的权重。INF 常量表示无穷大，用于初始化距离数组。代码如图4.3.1所示。

```
// 定义边的结构体
struct Edge {
    int to;
    int weight;
    Edge(int t, int w) : to(t), weight(w) {}
};

// 定义无穷大常量
const int INF = INT_MAX;
```

图4.3.1 定义边的结构体和无穷大常量

2. 初始化：dist 数组用于存储从源点到每个顶点的最短距离，初始时将所有顶点的距离设为无穷大，源点的距离设为 0。pq 是一个优先队列（小顶堆），用于存储 (距离，顶点) 对，每次从队列中取出距离最小的顶点进行处理。代码如图4.3.2所示。

```
    // 初始化距离数组，将所有顶点的距离设为无穷大
    vector<int> dist(n + 1, INF);
    // 源点的距离设为 0
    dist[s] = 0;

    // 定义优先队列（小顶堆），存储 (距离, 顶点) 对
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    // 将源点及其距离加入优先队列
    pq.push({0, s});
```

图4.3.2 初始化代码

3. 迭代过程：当队列不为空时，主循环不断从优先队列中取出距离最小的顶点进行处理；当队列为空时，满足终止条件，算法结束，此时 dist 数组中存储的就是源点到每个顶点的最短距离。

代码如图4.3.3所示：

```
while (!pq.empty()) {
        // 取出当前距离最小的顶点
        int currentDist = pq.top().first;
        int currentVertex = pq.top().second;
        pq.pop();

        // 如果当前距离大于已记录的距离，跳过
        if (currentDist > dist[currentVertex]) continue;

        // 遍历当前顶点的所有邻接边
        for (const Edge& edge : graph[currentVertex]) {
            int neighbor = edge.to;
            int weight = edge.weight;
            // 计算通过当前顶点到达邻接顶点的新距离
            int newDist = dist[currentVertex] + weight;

            // 如果新距离小于已记录的距离，更新距离并加入优先队列
            if (newDist < dist[neighbor]) {
                dist[neighbor] = newDist;
                pq.push({newDist, neighbor});
            }
        }
    }
```

图4.3.3 迭代过程代码

4. 主函数调用：主函数首先读取顶点数 n、边数 m 和源点 s。初始化图的邻接表 graph，并读取每条边的信息。调用 dijkstra 函数计算从源点到所有顶点的最短路径。输出结果，即从源点到每个顶点的最短距离。代码如图4.3.4所示。

```
int main() {
    int n, m, s;
    // 读取顶点数、边数和源点
    cin >> n >> m >> s;

    // 初始化图的邻接表
    vector<vector<Edge>> graph(n + 1);

    // 读取每条边的信息
    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        graph[u].emplace_back(v, w);
    }

    // 调用 Dijkstra 算法计算最短路径
    vector<int> dist = dijkstra(n, s, graph);

    // 输出结果
    for (int i = 1; i <= n; ++i) {
        cout << dist[i];
        if (i < n) cout << " ";
    }
    cout << endl;

    return 0;
}
```

图4.3.4 主函数调用代码

时间复杂度分析

基于堆优化的Dijkstra算法有两个阶段——**初始化阶段**和**迭代阶段**，接下来将详细说明这两个阶段的时间复杂度。

初始化阶段又分为距离数组初始化和优先队列初始化。距离数组初始化需要将源点到自身的距离设为 0，到其他顶点的距离设为无穷大。这个操作需要遍历所有的 V 个顶点，因此时间复杂度为 O(V)。优先队列初始化将源点及其距离（0）插入到优先队列中，插入操作在小顶堆中的时间复杂度为 O(log V)，但由于只插入一个元素，所以这部分时间复杂度可近似看作常数时间 O(1)。综合来看，初始化阶段的时间复杂度主要由距离数组初始化决定，为 O(V)。

迭代阶段也分两阶段：取出最小元素、更新距离并插入元素。在每次迭代中，需要从优先队列中取出距离最小的顶点。优先队列的取出最小元素操作的时间复杂度为 O(log V)。而整个算法最多需要进行 V 次取出操作（因为每个顶点最多被取出一次），所以这部分的总时间复杂度为 O(V log V)。对于每个顶点，需要遍历其所有的邻接边来更新相邻顶点的距离。图中所有顶点的邻接边总数为 E。当发现通过当前顶点到达某个相邻顶点的距离更短时，需要更新该相邻顶点的距离，并将其插入到优先队列中。插入操作在小顶堆中的时间复杂度为 O(log V)。因此，更新距离并插入元素这一操作的总时间复杂度为 O(E log V)。

将初始化阶段、取出最小元素和更新距离并插入元素的时间复杂度相加，得到：**O(V)+O(V log V)+O(E log V)**。当 V 和 E 足够大时，O(V)相对于 O(V log V) 和 O(E log V) 可以忽略不计。因此，基于堆优化的 Dijkstra 算法的总时间复杂度为 O((V + E) log V)。

4.3.2 单源最短路径算法——Bellman-Ford算法

算法简介

刚刚我们已经掌握了 Dijkstra 算法用于求解非负权图的最短路径问题。但当图中出现负权边时，Dijkstra算法便不再适用。此时，**Bellman-Ford算法**脱颖而出，它不仅能够处理带有负权边的图，还能检测图中是否存在负权回路。接下来，我们将深入学习这一重要算法。

Bellman-Ford算法（贝尔曼-福特算法）基于**松弛操作（relaxation operation）**，通过对图中所有边进行多次迭代更新，逐步逼近从源点到各个顶点的最短路径。其核心思想是：每一次迭代都尝试通过其他顶点来更新到目标顶点的距离，如果找到更短的路径，就更新该顶点的距离值。经过V - 1次迭代后（V为图中顶点的数量），若没有再发生距离更新，则说明已经找到了从源点到所有可达顶点的最短路径；若在V - 1次迭代后仍能更新距离，那就意味着图中存在负权回路。

算法详解

1. 初始化

给定一个有向图G=(V, E)，其中V是顶点集合，E是边集合，指定源点为s。创建一个数组dist，用于存储源点到每个顶点的最短距离估计值，初始时，将源点到自身的距离设为0，即dist[s] = 0，将其他所有顶点到源点的距离设为无穷大。

对于图中的每条边(u, v, w)（表示从顶点u到顶点v，边权为w），不进行任何操作，等待后续迭代更新。

2. 迭代过程

进行V - 1次迭代，这里的V是图中顶点的总数。每次迭代都遍历图中的每一条边(u, v, w)。

对于每一条边，尝试进行松弛操作。具体来说，如果dist[u] + w < dist[v]（即通过顶点u到达顶点v的距离比当前记录的dist[v]更短），则更新dist[v] = dist[u] + w。这一步的意义在于，当发现通过其他顶点中转到v的路径更短时，就用这个更短的路径距离来更新v的距离估计值。

3. 检查负权回路

在完成V - 1次迭代后，再进行一次边的遍历。

若在这次遍历中，仍然存在某条边(u, v, w)满足dist[u] + w < dist[v]，则说明图中存在负权回路。因为如果不存在负权回路，经过V - 1次迭代后，所有顶点的最短距离应该已经确定，不会再出现更短的路径。如果检测到负权回路，就说明从源点到某些顶点的最短路径是不存在的（因为可以沿着负权回路无限减小路径长度）。

在掌握了Bellman-Ford 算法的基本原理和操作步骤后，我们通过一个经典竞赛题目来加深对算法的理解与应用。

拓展知识

Johnson最短路

Johnson 算法的核心思想是通过重新赋权的方式，将一个可能包含负权边的图转换为一个不包含负权边的图，然后对每个顶点使用 Dijkstra 算法来求解最短路径。具体来说，它引入了一个虚拟源点，使用 Bellman - Ford 算法从该虚拟源点出发计算到其他所有顶点的最短路径，利用这些最短路径来对原图的边进行重新赋权，使得新图中所有边的权值都变为非负，这样就可以使用高效的 Dijkstra 算法来计算。

例题分析

【题目】

给定一个 n 个点的有向图，请求出图中是否存在从顶点 1 出发能到达的负环。（负环的定义是：一条边权之和为负数的回路。）

【输入格式】

本题单测试点有多组测试数据。输入的第一行是一个整数 T，表示测试数据的组数。对于每组数据的格式如下：第一行有两个整数，分别表示图的点数 n 和接下来给出边信息的条数 m。接下来 m 行，每行三个整数 u,v,w。若 w≥0，则表示存在一条从 u 至 v 边权为 w 的边，还存在一条从 v 至 u 边权为 w 的边。若 w<0，则只表示存在一条从 u 至 v 边权为 w 的边。

【输出格式】

对于每组数据，输出一行一个字符串，若所求负环存在，则输出 YES，否则输出 NO。

【分步解析】

1. 定义边的结构体：定义了一个名为 Edge 的结构体，用来表示图中的边。该结构体包含三个成员变量：

from：表示边的起始顶点。

to：表示边的终止顶点。

weight：表示边的权值。

结构体的构造函数 Edge(int f, int t, int w) 用于初始化这三个成员变量。

```
// 定义边的结构体
struct Edge {
    int from;
    int to;
    int weight;
    Edge(int f, int t, int w) : from(f), to(t), weight(w) {}
};
```

图4.3.5 定义边的结构体

2. 初始化距离数组

dist 是一个大小为 n + 1 的向量，用于存储从源点到各个顶点的最短距离估计值。初始时，将所有顶点的距离估计值设为 INT_MAX（表示无穷大）。因为源点是顶点 1，所以将 dist[1] 设为 0。代码如图4.3.6所示。

```
vector<int> dist(n + 1, INT_MAX);
dist[1] = 0;   // 源点为顶点1
```

图4.3.6 初始化距离数组

3. 进行 n - 1 次迭代松弛

外层循环 for (int i = 0; i < n - 1; ++i) 控制迭代次数，一共进行 n - 1 次迭代，这里的 n 是图中顶点的数量。在一个具有 n 个顶点的图中，任意两个顶点之间的最短路径最多包含 n - 1 条边，所以进行 n - 1 次迭代就可以找到最短路径。

内层循环 for (const Edge& e : edges) 遍历图中的每一条边。

条件判断 if (dist[e.from] != INT_MAX && dist[e.from] + e.weight < dist[e.to]) 的作用是：首先检查源点到边的起始顶点的距离是否已经确定（不为无穷大），然后比较通过当前边到达终止顶点的距离是否比之前记录的距离更短。如果满足条件，则更新 dist[e.to] 的值。

代码如图4.3.7所示：

```
// 进行n - 1次迭代松弛
for (int i = 0; i < n - 1; ++i) {
    for (const Edge& e : edges) {
        if (dist[e.from] != INT_MAX && dist[e.from] + e.weight < dist[e.to]
{
            dist[e.to] = dist[e.from] + e.weight;
        }
    }
}
```

图4.3.7 进行 n - 1 次迭代松弛

4. 检查负环

在完成 n - 1 次迭代后，再进行一次边的遍历。

如果存在某条边 e，使得 dist[e.from] + e.weight < dist[e.to]，这就意味着在经过 n - 1 次迭代后，仍然可以找到一条更短的路径，说明图中存在负环，函数返回 true。

如果遍历完所有边都没有发现这样的情况，说明图中不存在负环，函数返回 false。代码如图4.3.8所示。

```
// 检查负环
for (const Edge& e : edges) {
    if (dist[e.from] != INT_MAX && dist[e.from] + e.weight < dist[e.to]) {
        return true;  // 存在负环
    }
}

return false;  // 不存在负环
```

图4.3.8 检查负环

5. 主函数处理多组测试数据

首先读取测试数据的组数 T。

对于每组测试数据：读取顶点数 n 和边数 m。读取每条边的信息，将其存储在 edges 向量中。如果边的权值 w 大于等于 0，则添加一条反向的边，以处理双向边的情况。

调用 Bellman-Ford 函数检测图中是否存在负环，并根据返回结果输出YES或NO。

代码如图4.3.9所示：

```cpp
int main() {
    int T;
    cin >> T;  // 测试数据组数

    while (T--) {
        int n, m;
        cin >> n >> m;  // 顶点数n和边数m

        vector<Edge> edges;
        for (int i = 0; i < m; ++i) {
            int u, v, w;
            cin >> u >> v >> w;
            edges.push_back(Edge(u, v, w));
            if (w >= 0) {
                edges.push_back(Edge(v, u, w));  // 添加双向边
            }
        }

        if (bellmanFord(n, edges)) {
            cout << "YES" << endl;
        } else {
            cout << "NO" << endl;
        }
    }

    return 0;
}
```

图4.3.9 主函数处理多组测试数据

思考辨析

为什么Dijkstra在负权图上不可用呢？

Dijkstra 算法基于贪心思想，在处理负权图时会出现问题，主要原因有以下两点：

1、无法保证找到全局最优解

①Dijkstra 算法在运行过程中，一旦一个顶点被标记为已访问，就认为找到了从源点到该顶点的最短路径，不会再对其进行更新。

②但在负权图中，后续可能会通过负权边到达已访问的顶点，从而得到更短的路径。如果不重新考虑这些已访问的顶点，就无法找到全局最优解。

2、可能陷入无限循环

①当图中存在负权回路（即回路中边的权值之和为负数）时，Dijkstra 算法可能会陷入无限循环。

②因为负权回路会使总权值不断减小，算法可能会不断尝试沿着负权回路更新路径，导致无法终止。

4.3.3 全局最短路径——Floyd-Warshall算法

算法简介

在图论的众多算法中，最短路径问题一直是核心问题之一。此前我们学习了 Dijkstra 算法用于求解单源最短路径问题（从一个特定源点到其他所有顶点的最短路径），以及 Bellman - Ford 算法用于处理包含负权边的单源最短路径问题和检测负权回路。然而，在某些实际场景中，我们可能需要求解图中任意两个顶点之间的最短路径，这就是全局最短路径问题。Floyd - Warshall 算法便是专门用于解决此类问题的经典算法。

Floyd - Warshall 算法基于**动态规划**的思想。其核心原理是通过不断引入中间顶点，尝试更新任意两个顶点之间的最短路径。具体来说，对于图中的任意两个顶点 i 和 j，我们会依次考虑是否可以通过某个中间顶点 k 来缩短它们之间的距离。如果通过顶点 k 中转，即从 i 到 k 再到 j 的距离比当前记录的从 i 到 j 的距离更短，那么就更新这个最短距离。

算法详解

1. 初始化

给定一个有 n 个顶点的图 G=(V, E)，其中 V 是顶点集合，E 是边集合。创建一个 n * n的二维数组 dist，用于存储任意两个顶点之间的最短距离。

对于数组 dist，如果顶点 i 到顶点 j 有直接相连的边，且边的权值为 w，则 dist[i][j] = w；如果 i = j，则 dist[i][j] = 0；如果顶点 i 到顶点 j 没有直接相连的边，则 dist[i][j] = ∞。

2. 迭代更新

进行 n 次迭代，每次迭代选择一个中间顶点 k（k 从 1 到 n）。

对于每一对顶点 i 和 j（i 从 1 到 n，j 从 1 到 n），检查是否可以通过顶点 k 来缩短从 i 到 j 的距离。如果 dist[i][k] + dist[k][j] < dist[i][j]，则更新 dist[i][j] = dist[i][k] + dist[k][j]。

3. 结果判断

经过 n 次迭代后，数组 dist 中存储的就是任意两个顶点之间的最短距离。

同时，还可以通过检查 dist[i][i]是否为负数来判断图中是否存在负权回路。如果存在某个 i 使得 dist[i][i] < 0，则说明图中存在负权回路。

例题分析

【题目描述】

给出一张由 n 个点 m 条边组成的无向图，求出所有点对 (i,j) 之间的最短路径。

【输入格式】

第一行为两个整数 n,m，分别代表点的个数和边的条数。接下来 m 行，每行三个整数 u,v,w，代表 u,v 之间存在一条边权为 w 的边。

【输出格式】

输出 n 行每行 n 个整数。第 i 行的第 j 个整数代表从 i 到 j 的最短路径。

【分步解析】

1. 初始化矩阵距离

创建一个 n × n 的二维向量 dist 作为距离矩阵，用于存储任意两点之间的最短路径长度。初始时，将所有元素设为 INF（这里使用 INT_MAX / 2 避免后续相加时溢出），表示两点之间初始距离为无穷大，即不可达。

对于矩阵的对角线元素 dist[i][i]，将其设为 0，因为一个点到自身的距离显然为 0。代码如图4.3.10所示。

```
vector<vector<int>> dist(n, vector<int>(n, INF));
for (int i = 0; i < n; ++i) {
    dist[i][i] = 0;
}
```

4.3.10 初始化矩阵距离

2. 读取边的信息并更新距离矩阵

读取 m 条边的信息，每条边包含起点 u、终点 v 和边权 w。由于输入的点编号通常从 1 开始，而代码中数组索引从 0 开始，所以将 u 和 v 减 1。

对于无向图，边是双向的，所以同时更新 dist[u][v] 和 dist[v][u] 的值，取当前值和新边权 w 中的较小值。代码如图4.3.11所示。

```
for (int i = 0; i < m; ++i) {
    int u, v, w;
    cin >> u >> v >> w;
    u--; v--;
    dist[u][v] = min(dist[u][v], w);
    dist[v][u] = min(dist[v][u], w);
}
```

4.3.11 读取边的信息并更新距离矩阵

3. 核心的 Floyd - Warshall 算法部分

最外层循环的变量 k 表示中间节点，它尝试所有可能的中间节点。内层的两层循环 i 和 j 遍历所有的点对。

对于每一个点对 (i, j)，检查是否可以通过中间节点 k 使得从 i 到 j 的路径更短。如果 dist[i][k] 和 dist[k][j] 都不是无穷大（即 i 到 k 和 k 到 j 都可达），并且 dist[i][k] + dist[k][j] 小于当前记录的 dist[i][j]，则更新 dist[i][j] 的值为 dist[i][k] + dist[k][j]。代码如图4.3.12所示。

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] +
dist[k][j] < dist[i][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

4.3.12 核心算法部分

4. 输出结果

遍历距离矩阵 dist，如果 dist[i][j] 仍然是 INF，说明从 i 到 j 不可达，输出 INF；否则输出 dist[i][j] 的值。代码如图4.3.13所示。

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (dist[i][j] == INF) {
            cout << "INF ";
        } else {
            cout << dist[i][j] << " ";
        }
    }
    cout << endl;
}
```

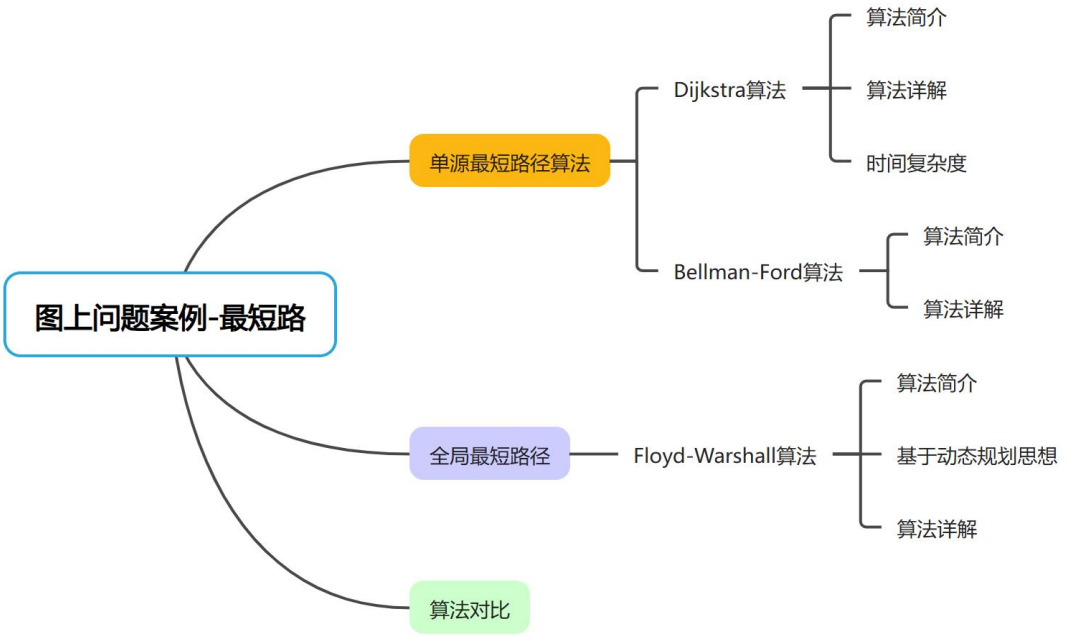
4.3.13 输出结果

知识链接：算法对比

我们已经学习了基于堆优化的 Dijkstra 算法、Bellman - Ford 算法和 Floyd - Warshall 算法。这三种算法各有特点，在不同的场景下有着不同的表现。下面我们将详细对比它们的时间复杂度和适用场景。

算法	时间复杂度	使用场景
基于堆优化的 Dijkstra 算法	O((V + E)log V)	非负权图的单源最短路径问题，稀疏图
Bellman - Ford 算法	O(VE)	包含负权边的单源最短路径问题，检测负权回路
Floyd - Warshall 算法	O(V^3)	全局最短路径问题，顶点数较少的图

知识图谱



练习提升

1. 某外汇交易平台提供货币兑换汇率表。例如，1美元兑换0.9欧元，1欧元兑换110日元，1日元兑换0.01美元。设计一个算法，判断是否存在通过循环兑换获利的可能（即负权环）。

【输入】

货币种类数 n 和兑换关系数 m。接下来 m 行，每行为货币A与货币B汇率，表示1单位A可兑换的B的数量。

【输出】

“存在套利机会” 或 “无套利机会”。

2. 在社交网络中，若两个人的最短好友链长度 ≤6，则称他们满足“六度空间”理论。给定好友关系图（无向无权图），计算所有点对中最短路径的最大值，验证是否 ≤6。

【输入】

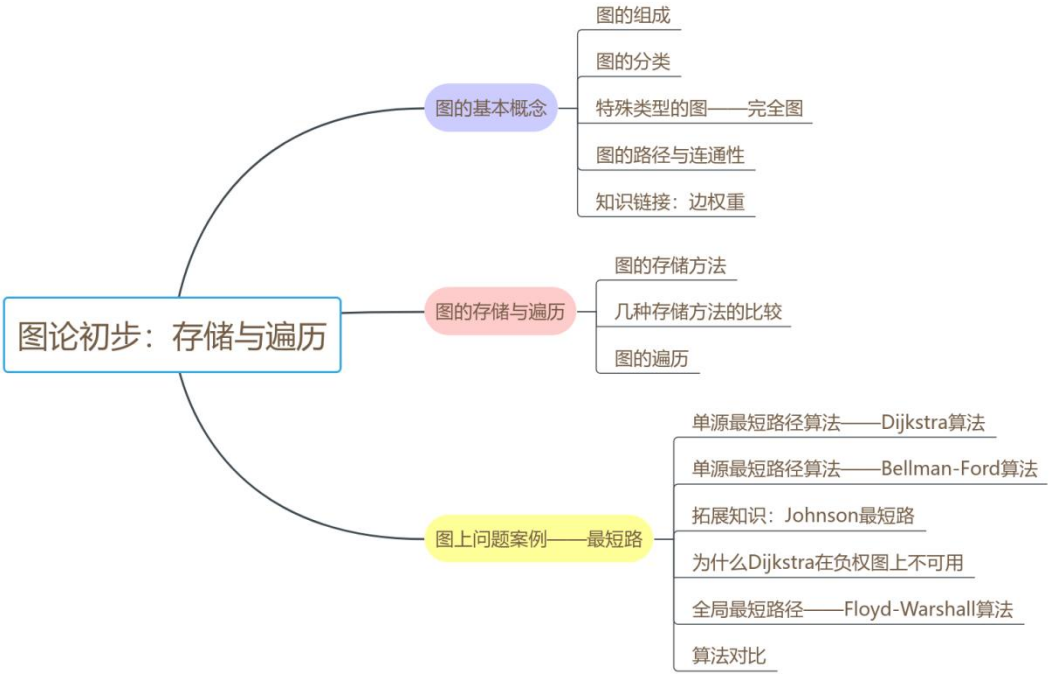
用户数 n 和好友关系数 m。接下来 m 行，每行为 u v，表示用户 u 和 v 是好友。

【输出】

最长的最短路径长度。若>6，输出“不符合六度空间”，否则输出“符合”。

总结 评价

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
图的基本概念与术语	□不了解 □了解 □理解
图的存储结构	□不了解 □了解 □理解
图的遍历算法	□不了解 □了解 □理解
最短路径算法	□不了解 □了解 □理解
负权环检测	□不了解 □了解 □理解
完全图与稀疏图的特点及适用场景	□不了解 □了解 □理解
图的动态操作	□不了解 □了解 □理解

小结

在本章的学习过程中，我们首先深入探索了图的基本概念。从顶点、边、权值这些构成图的核心元素出发，我们认识到，顶点就像是现实世界中的一个个实体，边则代表着实体之间的关系，而权值能够量化这些关系的某些属性。通过对无向图和有向图等不同类型图的特点分析，我们明白了图结构在描述不同关系时的灵活性和多样性，这是构建图论知识体系的重要基石。

随后，我们聚焦于图的存储方式，深入学习了邻接矩阵和邻接表这两种重要方法。邻接矩阵以二维数组的形式，直观地展现了顶点之间的连接关系，在判断两点是否相连时效率极高，但是在处理稀疏图时，会造成大量的空间浪费；邻接表则通过链表或动态数组的方式，将与每个顶点相连的边存储起来，这种方式在存储稀疏图时优势明显，能够有效节省空间。我们明白了，根据实际问题的特点和数据规模，合理选择存储结构，是提升程序运行效率的关键。

在图的遍历环节，广度优先搜索（BFS）和深度优先搜索（DFS）为我们提供了两种截然不同却又同样高效的探索方式。BFS 按照距离起始顶点由近到远的顺序访问节点，常用于寻找最短路径等问题；而 DFS 则沿着一条路径不断深入，直到无法继续才回溯，这种方式在寻找连通分量等问题上表现出色。掌握它们的逻辑与实现方法，我们就拥有了在图结构中自由穿梭、获取数据的有效途径。

最后，我们通过最短路问题这个经典案例，将前面所学的理论知识应用于实践。迪杰斯特拉（Dijkstra）算法和弗洛伊德（Floyd）算法的学习与实践，让我们深刻体会到了图论算法的精妙之处。Dijkstra 算法适用于边权非负的图，通过不断选择距离源点最近的未确定顶点，逐步扩展最短路径；Floyd 算法则基于动态规划思想，能够一次性求出图中任意两点之间的最短路径。这些算法不仅帮助我们解决了具体的问题，更让我们学会了如何将理论转化为实际的代码实现。

图论知识不仅是 CSP-J 竞赛的重要考点，更是解决复杂实际问题的有力武器。它教会我们用计算机的思维，将现实世界中错综复杂的关系抽象成能够处理的数据结构，通过算法优化找到最佳解决方案。通过这一章的学习，希望同学们不仅熟练掌握图论的基本概念和算法，更要注重培养抽象建模和逻辑思维能力。